Identifying Extract Method Refactorings

Omkarendra Tiwari and Rushikesh K. Joshi
{omkarendra,rkj}@cse.iitb.ac.in
Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai, India

ABSTRACT

Extract method refactoring identifies and extracts a set of statements implementing a specific functionality within a method. Its application enhances the structure of code and provides improved readability and reusability. This paper introduces Segmentation, a new approach for identifying extract method opportunities focusing on achieving higher performance with fewer suggestions. Evaluation of the approach includes six case studies from the open-source domain, and performance is compared against two state-of-the-art approaches. The findings suggest that Segmentation provides improved precision and F measure over both the approaches. Further, improved performance is reflected over long methods too.

KEYWORDS

Extract Method Refactoring, Long Methods, Modularity, Restructuring, Segmentation, Segment Graph

ACM Reference Format:
Omkarendra Tiwari and Rushikesh K. Joshi. 2022. Identifying Extract Method Refactorings. In *15th Innovations in Software Engineering Conference (ISEC 2022), February 24–26, 2022, Gandhinagar, India.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3511430.3511435

1 INTRODUCTION

Extract method is known to be a suitable refactoring solution for various code smells such as *long method*, *code clone*, and *feature envy*[6]. In a study, Kim et al. [9] noted that readability, testability, reusability, adding new features, bug fixing are leading causes for refactoring. Further, these causes were also reported as primary motivations for extract method refactoring in a study by Silva et al. [19]. Murphy-Hill et al. [17] observed that extract method refactoring tools are seldom used, and instead, such refactoring is carried out manually.

We note that recent approaches [5, 18, 21] provide a large number of extract method opportunity (EMO) suggestions, which may result in a high recall but with a lowered precision. Also, due to overlapping, multiple suggestions refer to the same functionality. Given the various applications of extract method refactoring, it can benefit practitioners if the computed suggestions can be used to gain further insight into the structure of the native method under refactoring, in addition to identifying promising functionalities for extraction [20].

In this paper, we present *Segmentation*, a novel approach aiming to achieve higher precision and comparable recall with comparatively fewer suggestions. The Segmentation algorithm computes EMO suggestions by analyzing blocks of statements in the bottom-up and innermost-first manner. This order of processing allows the approach to first identify the core of functionality and then merge the contributing statements with functional coherence to computations in the block. These include statements that supply data to the block for computing the required output or consuming the generated output(s) of the block.

Segmentation generates disjoint suggestions. Disjoint suggestions restrict the number of identified functionalities, as well as they, allow the user to gain further insight into the method under refactoring. For example, since they are disjoint, the count of suggestions may be used to detect the *long method* code smell, and further, the interdependence among identified functionalities can be used for computing the smell severity of the detected long method code smell. Such insights can be crucial as various studies reported that often basic refactorings are part of larger structural enhancements aimed at improving existing software [9, 22].

Further, to assess the usefulness of the approach and identified suggestions, empirical validation is provided over six open-source applications. Among them, JUnit, JHotDraw, and MyWebMarket are three Java-based applications, and these have earlier been used as a benchmark in the works of Silva et al. [18], and Charalamidou et al. [5]. The third open-source application used is the XData system [4], which assists in grading answers that are SQL queries in a learning environment. The inclusion of the XData application in our comparative analysis helped evaluate the scalability of the approaches since all its method sizes range between 30 to 299 LoC, constituting relatively large method bodies. Additional two studies considered are EventBus and Mockito, where EMOs are synthetically created by inlining method calls using the guidelines provided by Silva et al. [18].

The performance of the approach is compared with two state-of-the-art tools, namely, JExtract [18] and SEMI [5]. The results suggest that Segmentation identifies extract method opportunities with comparably high precision with small-sized methods and long methods. Also, despite JExtract leading in the recall, Segmentation outperforms both the approaches in F-measure.

The remainder of this paper is structured as follows. Section 2 discusses the identification and ranking of extract method opportunities and demonstrates the approach through an example. An evaluation of the approach is presented in Section 3. Related work in recent years is reviewed in Section 4.

2 SEGMENTATION

As mentioned in the previous section, the motivation for developing the proposed approach is not limited to achieving higher performance with fewer suggestions but also to extend the applicability of the results in gaining further insights about the method. In this section, we describe the design decisions made to complement the goals of Segmentation. Before exploring the details, we outline Segmentation approach for identifying a functionality (EMO) within a method.

(1) Clustering related statements
   (a) Select a list of attributes to classify two statements as related, and
   (b) Devise a method to form external behavior preserving clusters of related statements.
(2) Identifying EMOs
   (a) Define a fitness function to classify/rank clusters.

In a nutshell, *Segmentation* utilizes already clustered statements present in the form of block-statements (if-else, iteration, try-catch etc.) to identify a core functionality and then includes supporting statements to the core functional cluster. Finally, the obtained cluster is analyzed for classifying it as an EMO via a fitness metric. Subsequent subsections discuss the identification and ranking of EMOs.

2.1 Forming Clusters of Related Statements

A common identifier is a widely used attribute among clustering approaches for identifying related statements. Segmentation is no exception, but it also considers the role of a common identifier in two statements for classifying them as related. This similarity measure called as *define-use* association helps the approach prevent the generation of a large count of clusters. Further, to quickly identify core-functionality, another similarity measure *Block association* is proposed. The process of identifying related statements using both the associations and then clustering is discussed now.

2.1.1 Identifying Related Statements. Segmentation identifies related statements based on two kinds of associations namely, *block* and *define-use* association. Any two statements exhibiting either association are marked related and are further analyzed if they can be clustered. The two associations are defined as follows:

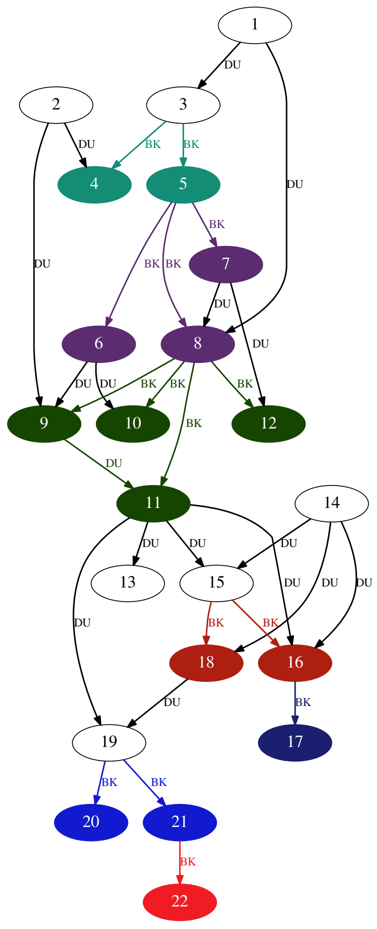
- *Block association*: A set of statements constituting the body of a block is said to be in *block-association* with the entry statement of the block. Entry statement of a block often constitutes a branching statement such as if, else, elseif, for, while, try, catch, switch, or case-statement. It can be noted that *Segmentation* models all the blocks (if/loop/switch/try etc.) as non-branching when a source code is transformed to a *Segment Graph*, an intermediate representation over which *segmentation algorithm* is applied to cluster related statements.
  Segmentation abstracts away branching details as it considers already clustered statements (such as conditional or iterative blocks) as indicators of functionally coherent clusters and utilizes their association with other individual or block statements to identify an auxiliary functionality.
  Example: Consider the program for Fibonacci prime shown in Figure 1(a). It contains various conditional and iterative blocks. One such block starts at line 8 and spans up to line 12. Thus, statements 9-12 are in *block association* with statement 8, which is an entry statement of the block. Similarly, statements 6-12 are in *block association* with statement 5.
- *Define-use association*: Statements S_i and S_k are in *define-use* association via variable x if the former defines variable x and the latter uses its value, provided that no statement S_j defines variable x, where $i < j < k$.
  Example: Consider statements 7 and 8, the former defines a variable 'i', which is used by the latter. Thus, these two are in *define-use* association. Similar is the case with 6 and 9.
  Consider S_1, S_2,...,S_n be a set of statements such that S_i is in *define-use* association with only S_{i+1} then the set is called as *chain* of *define-use* association, where $0 < i < (n - 1)$.

2.1.2 Clustering Related Statements. Segmentation aims at forming clusters that are external behavior preserving. For this purpose, after a statement is classified as related, it is included in a cluster only if it satisfies the corresponding association clustering condition. Clustering conditions for both associations are discussed below.

- *Block Association Clustering (BAC):* Statements classified as related with the entry statement of a block require to satisfy following two conditions for clustering.
  (1) Block statements can be clustered with only a primary block entry statement. Blocks beginning with *if*, *for*, *while*, *try*, *switch* etc represent primary blocks, whereas *else*, *elseif*, *catch* etc are non-primary blocks.
  (2) An entry statement is always clustered with all the statements of the block, including nested blocks or non-secondary blocks associated with the block.
  Example: Consider the statement 9. It belongs to a block with entry statement 8, and thus, it is in *block* association with entry statement 8. But it can not be clustered with the entry statement alone as condition 2 is not satisfied. Although statements 9 to 12 can be clustered with entry statement 8, and thus, forming a valid cluster constituting statements 8-12.
- *Define-use Association Clustering (DAC):* A pair of statements, say S_i and S_j, classified as related via *define-use* association can be clustered only if the following conditions are satisfied.
  (1) Both the statements belong to the same block.
  (2) Statement defining the common variable must be in *define-use* association with exactly one statement/cluster.
  (3) Statement S_i is not anti dependent on Statement S_k, where $i < k < j$.
  Example: Consider the statement 2 and 4. They can be qualified as related via *define-use* association on variable 'a'. But they can not form a cluster as statement 4 belongs to block 3, which is not the case with statement 2. Although, if the block at statement 3 is clustered, then statement 2 would be in *define-use* association with statement/cluster 3 and statement 2 can be included into cluster at statement 3. The resulting cluster then would have statements 2-12.

```
void FiboPrime()        {
  int i, n, a, b, t;
0.  printf("Value␣of␣N:");
1.  scanf("%d",&n);
2.  a = 0;
3.  if (n ==1)
4.   printf("NthTerm:%d",a);
5.  else       {
6.   b = 1;
7.   i = 3
8.   while (i <=n){
9.    t = a + b;
10.   a = b;
11.   b = t;
12.   i++;
    }
    }
13. printf("NthTerm:%d",b);
14. i=2;
15. for (; i<=b/2;){
16.  if (b%i == 0)
17.    break;
18.  i++;
    }
19. if (b<=1 || i <=b/2)
20.  printf("Not␣Prime");
21. else
22.  printf("Prime");
  }
```

(a) Source Code



(b) Segment Graph

Figure 1: Source Code of Fibonacci code and corresponding segment graph (Intermediate Represenation)

In the case of *chain* of related statements via *define-use* association, all statements of the chain that are in the same block can form a cluster.

2.2 Identifying Extract Method Opportunity

Previously mentioned clustering methods can provide an EMO around a selected statement. For example, the application of BAC on statement 3 would produce a cluster of statements 3-12. And the further application of DAC on statement/cluster 3 would result in the inclusion of statement 2 to the cluster. This EMO contains the functionality for computing the Nth term of the Fibonacci series. Although, the application of the same combination of clustering on statement 8 would result in an undesired cluster as it does not contain statements from the outer if-else block. Hence, this EMO would include partial functionality.

To automatically identify promising EMOs, Segmentation uses two fitness metrics, namely *Lack of Computational Strength* (LoCS) and *Parent Affinity* (PA). LoCS aims at classifying a cluster of statements as a core-functional block. It analyses data sharing among statements within and outside the cluster. Once a cluster is classified as a core-functional block, PA utilizes the data sharing present between two consecutive blocks to determine if both the blocks contribute to the same functionality. The blocks are analyzed in bottom-up order that is from innermost to outermost block. Once a cluster is marked as an EMO, all the statements in it are excluded from further consideration for clustering. This way, Segmentation keeps all suggested EMOs disjoint and thus, lowering the total number of suggestions.

2.2.1 Segmentation Algorithm

2.2.1 Segmentation Algorithm . A high-level view of *Segmentation* approach for identifying EMOs is presented in Algorithm 1. The algorithm is applied over a graph-based intermediate representation of input source code, called *Segment Graph*. The vertices in the graph represent statements of the method under process, and edges capture association among statements. Segmentation analyses primary blocks in bottom-up order to identify if the statements of the block constitute a functionality (steps 1-3). Each block is analyzed, and the LoCS metric is computed; if the LoCS measure for the block is below the specified threshold, the block is marked as a candidate EMO (steps 4-8).

Further, the algorithm analyses associated statements of the identified core-functionality block to cluster statements contributing to its functionality. Two kinds of contributing statements are: (i) an outer block containing the core-functional block, and (ii) statements either supplying or consuming data to/from the core-functional cluster. PA metric measures the functional similarity between two consecutive blocks to identify the first kind of contributing statements. For identifying second kinds of statements, define-use association (steps 9-16) is applied.

Once a block is clustered with all its contributing statements, it is marked as an EMO. Such marked clusters are excluded from further consideration in the EMO identification process. This way, Segmentation achieves disjoint EMO suggestions.

Algorithm 1 Segmentation: Identifying EMOs

```
1:  $entryList \leftarrow primaryBlockEntryStatements(segGraph)$
2:  while $hasElements(entryList)$ do
3:      $current \leftarrow removeLast(entryList)$
4:      if $computeLoCSofBlock(current) < LoCSThreshold$ then
5:          $hasParent = TRUE$
6:          while $hasParent$ do
7:              $applyBACoverBlock(current, segGraph)$
8:              $applyDACoverCluster(current, segGraph)$
9:              $hasParent \leftarrow hasOuterBlock(current, segGraph)$
10:             $parent \leftarrow last(entryList)$
11:             $isCoherent \leftarrow computePA(parent) < PAThreshold$
12:             if $hasParent \wedge isCoherent$ then
13:                 $current \leftarrow removeLast(entryList)$
14:             else
15:                 $hasParent = FALSE$
16:             end if
17:         end while
18:         $markClusterasEMO(current, segGraph)$
19:     end if
20: end while
```

The following subsection describes the segment graph, construction of LoCS and PA metrics, and application of the algorithm over an example.

2.3 Fitness Metrics

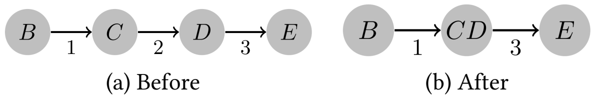
This section describes the formulation of LoCS and PA metrics in detail, along with an intermediate representation *Segment Graph*.

2.3.1 Segment Graph. It is a directed-graph-based representation of source code, where each vertex corresponds to exactly one statement. A vertex is assigned the same label as its corresponding source code statement. Labeled edges represent the association between statements. An edge $\langle u, v, DU \rangle$ reflects a *define-use* association between statement u and v due to presence of a common variable that is defined at the former and used at latter. *Block* association between statements is represented by edge $\langle u, v, BK \rangle$, where u represents block entry statement.

Example: Segment Graph of a source code for Fibonacci Prime is illustrated in Figure 1. Statements at index 4 and 5 are are in *block association* with entry statement at index 3 (if-statement), whereas statements 6, 7, and 8 are directly inside the block at statement 5 (else-statement). This *block association* is captured by a directed *block association* edges (shown as edges with label 'BK') $\langle 3, 4 \rangle$, $\langle 3, 5 \rangle$, $\langle 5, 6 \rangle$, $\langle 5, 7 \rangle$, $\langle 5, 8 \rangle$. In the figure, for the sake of clarity, all vertices sharing the same parent block have been shown in same color. For example, vertex 3 is parent block entry statement of 4 and 5, and vertex 9 is parent block entry statement of vertices 9-12. Similarly, *define-use* association edges $\langle 1, 3 \rangle$ and $\langle 1, 8 \rangle$ reflect the use of variable n at index 3 and 8, and its value assignment at index 1 in the code. Also, an edge $\langle 11, 13 \rangle$ reflects the consumption at line 13, of value generated at statement 11. This edge corresponds to variable b.

A few terminologies related to *Segment graph* are discussed.

- *Producer*: A statement S_i is called as producer if it is in *define-use* association with statement S_j, where $i < j$. *Producers* can be identified as vertices with at least one outgoing edge with label DU in a *segment graph*.
- *Relay*: A producer statement that is in *define-use* association with at least one statement outside the block is called as *relay*. For example, in Figure 1, source code statement 11 is relay for blocks 3,5 and 8.
- *Exclusive Source*: A vertex u is said to be an exclusive source to another vertex v, if u have no incoming DU edge and exactly one outgoing DU edge to v. Such a vertex represents a statement that defines a variable used by exactly one statement or block, and hence, functional coherence between the two is high. For example, statement/vertex 7 is an exclusive-source as it is in *define-use* association with statements/vertices 8 and 12, both belonging to block at 8. Thus, if the block at 8 is clustered, vertex seven would be an exclusive source to the cluster at 8.
- *Edge Contraction*: Edge contraction is an edge removal operation that results in merging of its two end vertices into a new vertex and mapping of all incident edges on merged vertices to the new vertex [7]. For example, in Figure 2 the edge connecting vertices C and D is contracted and the new vertex is named CD.



$$\begin{tikzcd} B \arrow[r, "1"'] & C \arrow[r, "2"'] & D \arrow[r, "3"'] & E \end{tikzcd} \qquad \begin{tikzcd} B \arrow[r, "1"'] & CD \arrow[r, "3"'] & E \end{tikzcd}$$

(a) Before (b) After

Figure 2: Contraction of Edge connecting C to D

- *Block Contraction*: Once a set of statements are marked as an EMO, the corresponding subgraph in *segment graph* is contracted to a single vertex, and all incoming/outgoing edges to/from the contracted subgraph are mapped to this remained vertex. Segmentation assigns the label of the resulting vertex the same as the outermost block entry vertex of the subgraph. For example, when block at 15 is contracted in Figure 4(a), the resulting shrunk *segment graph* shown in Figure 4(b) contains an edge from vertex 15 to 19 that is due to vertex 18.

2.3.2 *Configuring Segmentation.*

As Murphy-Hill et al. [17] noted that users often apply tools without altering default settings, we aim to provide a default value for which we observed promising results. Segmentation uses three parameters to regulated EMO identification. Two of these, LoCS and PA, are set by default to a value based on observation of results obtained by applying *Segmentation* to a set of input methods. The default threshold for LoCS metric is 0.41, and for PA it is 0.34. An inner block is first considered for contraction only if its LoCS value is below the threshold. Similarly, a parent block is considered to be closely associated with its inner block if the PA value is below 0.34. Now, we discuss fitness metrics for assessing the quality of clusters and decide if the identified functionality can be suggested as an EMO.

2.3.3 *Lack of Computational Strength (LoCS).*

Metric LoCS acts as a binary classifier and serves the following two purposes, (i) it restricts the number of suggestions to only promising candidates, (ii) it prevents extraction of smaller components of a large functionality. It measures the computational strength of a given block in terms of producer vertices that directly or indirectly contribute to relay vertices of the block. The formulation for LoCS is given below.

Let b_v be a block rooted at a primary block entry vertex v in segment graph G, and it is identified as a candidate auxiliary functionality. Let $Relay_G(b_v)$ be a set of all relay vertices in b_v. Boolean expression $Path_D(u, r, b_v, G)$ indicates if there exists a path composed of DU-edges from vertex u to vertex r in b_v. Function $Producer_G(b_v)$ returns the set of all producer vertices in b_v. Function $RelayShare_G(b_v, r)$ provides a set of all producer vertices in b_v from each of which, at least one relay vertex is reachable through a path composed only of DU-edges. In other words, $RelayShare_G(b_v, r) = \{u : u \in Producer_G(b_v), \exists r \in Relay_G(b_v) \wedge Path_D(u, r, b_v, G)\}$.

Even if a producer is not a member of $RelayShare_G(b_v)$, it is a part of computational unit of the block. The set of producer vertices that are not members of any *RelayShare* is represented by function $NonRelayShare_G(b_v)$. Count $\#TotalRelayShare$ represents the aggregate count of all the data supplies to relay vertices. Data supplies are DU-edges, not supplier vertices, since every supply is independently counted. Count $\#Relay$ is the cardinality of set $Relay_G(b_v)$.

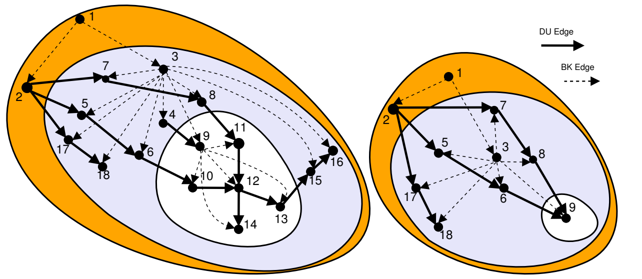
Lack of Computational Strength (LoCS) is computed as the ratio of number of relay vertices in a block to the aggregate computational strength of the block. The denominator is the count of total supplies associated (both internal and external) with the block. It is computed as the sum of $\#NonRelayShare$ and $\#TotalRelayShare$. Heavy sharing implies higher computational strength, which indicates lower values of LoCS. These quantifications are listed below.

$$AllRelayShare_G(b_v) = \bigcup_{r \in Relay_G(b_v)} RelayShare_G(b_v, r)$$

$$NonRelayShare_G(b_v) = Procedure_G(b_v) \setminus AllRelayShare_G(b_v)$$

$$TotalRelayShare_G(b_v) = \sum_{r \in Relay(b_v, G)} |RelayShare(b_v, r, G)|$$

$$LoCS_G(b_v) = \frac{|Relay_G(b_v)|}{TotalRelayShare_G(b_v) + |NonRelayShare_G(b_v)|}$$



(a) Original segment graph (b) Contraction of block at 9

Figure 3: A sample segment graph with nested Blocks

Attributes	Vertex Set	Count
Relay Nodes	{13}	1
Sink Nodes	{14}	1
Exc. Source	{4}	1
Producer Nodes	{4,10,11,12,13}	5
RelayShare	{10,11,12}	3
NonRelayShare	{4,13}	2
Incoming Chains	$\{\langle 5, 6\rangle, \langle 6, 10\rangle\}$ & $\{\langle 7, 8\rangle, \langle 8, 11\rangle\}$	2
Outgoing Chain	$\{\langle 13, 15\rangle, \langle 15, 16\rangle\}$	1

Table 1: Analysis of Block 9 in Figure 3

The lower the value of $LoCS$, the better is the possibility of extracting block b_v as a separate function. A block is accepted as an EMO only if (i) value of $LoCS_G(b_v)$ is less than a threshold (we use 0.41 as threshold which is obtained from experimentation), and (ii) its parent block (if it exists) contains a significant separate computation. Such computations may share data with inner block (i.e. b_v) in the form of parameters or return value.

Example Use of LoCS: An example segment graph is shown in Figure 3(a). It contains three nested blocks. In the figure dotted arrows represent block edges, and thick arrows represent DU-edges. It can be seen that the outermost block is rooted at vertex 1, middle block is rooted at vertex 3, and the innermost block is rooted at vertex 9. Table 1 lists the attributes associated with block 9. Since the block has at least one relay statement (vertex 13), it is considered as a candidate for extract method opportunity and its LoCS is computed. As the table shows, block 9 has four vertices with one or more outgoing DU-edges (10-13), one exclusive source vertex (vertex 4) and one relay vertex (vertex 13). Therefore, LoCS for the block is $\frac{1}{5}$, which is below 0.41, the default threshold used for this example. Hence, the block is contracted, and further, the exclusive source vertex and the outgoing chain (consisting of vertices 13,15,16) are merged to it. Figure 3(b) shows the segment graph after contraction of the block at vertex 9 (consisting of vertices 4 and 9 to 16). After contraction of this block, we need to investigate the parent block at vertex 3. The contraction of parent block is explained next.

2.3.4 *Parent Affinity.*

It measures the functional difference between two nested overlapping blocks, where the inner block is identified as a candidate opportunity, leading to its contraction into a single vertex. It is used to determine whether the inner block should be merged with the parent block and be extracted together as a single functionality.

For this purpose, the distinct functionality implemented in the parent block is quantified in terms of distinct computations (computations which are not associated with inner block via data exchange). Distinct computations are computations which do not supply or consume data to/from inner block. Based on the degree of coherence of computations the inner block with the parent block, four cases of merger possibility arise. These are enumerated below. It can be observed that in some of them, the decision is evident due zero overlapping. But in the case of overlapping, we use this metric to decide the possibility of merger of the inner block into its parent block. The decisions taken in each of these four cases are as follows: (i) Parent block has a distinct relay statement, and all its computations are connected with inner block: In this case they can be merged into one block due to its close association. (ii) Parent block has a distinct relay statement, and not all or none of its computations are connected with inner block: In this case, their merger possibility is decided by PA.(iii) Parent block has no distinct relay statement, and it is either empty or all its computations are connected with the inner block: They can be merged into one block due to its close association. (iv) Parent block has no distinct relay statement, and none or all its computations are connected with inner block is present: Their merger possibility is decided by Parent Affinity (PA). The metric *parent affinity* (PA) is defined as follows.

$$PA(v, G) = 1 - \frac{IndependentNodes(b_p, b_v, G)}{ParentDataNodes(b_p, b_v, G)}$$

where, (i) b_v and b_p represent the blocks respectively at block entry vertex v and its parent block entry vertex p, (ii) *ParentDataNodes* (b_p, b_v, G) is the count of those vertices in b_p from which there is at least one outgoing *DU*-edge. (iii) *IndependentNodes* (b_p, b_v, G) is the count of those vertices in b_p which are not directly connected with b_v by a *DU*-edge (after contraction of b_v).

Example Use of Parent Affinity: Figure 3 (b) shows the segment graph after contraction of the inner block rooted at vertex 9. After the contraction of this block, its parent block rooted at vertex 3 still contains six vertices in addition to the contracted vertex 9 and vertex 3 itself. Two of these vertices (6 and 8) are directly connected with a *DU*-edge to block 9, whereas, vertices 5 and 7 are indirectly connected to block 9 via *DU*-edges. In block 3, only two vertices (17 and 18) that do not supply/consume data to/from block 9 remain. Thus, $Producer(b_p, G)$ is the vertex set V = {5,6,7,8,17}. Out of these, producer vertices 5, 7, and 17 are not directly connected with inner block 9. They form the set *IndependentNodes*. So, parent affinity (PA) for block 3 is (1 - (3/5)) = 0.4. Since it is higher than default threshold of 0.34, this parent block is not merged with inner block 9. Thus, only inner block 9 with vertices 4, 9-16 is extracted as a separate method.

2.4 Example: Refactoring using Segmentation Algorithm

This section provides an overview of the process of refactoring using the *Segmentation* through an example of *Fibonacci Primes* as shown in Figure 1. The figure shows an example program and its corresponding *segment graph*. Each pair of statement related via *define-use* or *block* association is represented through vertices connected by a labeled edge. The edge label captures the association between two statements/vertices.

Since, *Segmentation* inspects all the blocks in reverse order of appearance, so that it can analyze and classify inner blocks as core of a functionality and then outer blocks can be clustered with it and suggested as an EMO. In our example, all primary blocks begin at 3, 8, 15, 16 and 19.

- *Block 19:* It is an *if-block*. It has two *block association* statements 20 and 21. Vertex 21 represents an entry statement for *else-block* that is in *block association* with statement 22. Thus, the block at 19 is spanned till statement 22 and it does not have any *define-use* association as there are no statement defining a variable. The same can be observed from the *segment graph* as block at 19 has no outgoing *DU*-edge connecting to any vertex outside of the block (i.e. after vertex 22). Hence, it is not considered as an extraction opportunity, and it is contracted into vertex 19 as shown in Figure 4(a).
- *Block 16:* Block rooted at vertex 16 (if-block) is analyzed next. Similar to block 19, this block too does not have any *producers*. Hence, it is not considered for extraction. The block is not contracted because it is not an outermost block.
- *Block 15:* It is an iterative block. It can be observed that it has an outgoing *DU*-edge from vertex 18, and thus, it has a *relay*. Hence, this block is further examined for assessing its extraction possibility. It has two producers one is 14 (exclusive source to the block) and 18. Lack of computational strength measure for this block happens to be 0.5, that is higher than a default threshold of 0.41. So, this block is not selected for extraction. As this block is not in *block-association* with any other block, it is contracted as shown in Figure 4 (b). Now, an exclusive define-use association on the contracted block can be observed in form of a *DU*-edge $\langle 14, 15 \rangle$. Such a association is identified by source vertex and such a define-use association between two vertices shows strong association between them. Hence, as the next step, vertex 14 is merged with vertex 15 by contracting edge $\langle 14, 15 \rangle$ as shown in Figure 4 (c).
- *Block 8:* Further, LoCS for block 8 is measured as it has one *relay*, vertex 11. Block 8 has two vertices (9, 11) with at least one outgoing *DU*-edge and two exclusive source vertices (6, 7). So, LoCS value for the block is 0.25, which is lower than the threshold 0.41. Hence, the block is qualified for suggesting as an EMO and it is contracted to a single vertex as shown in Figures 4 (d) and (e). Now, its outer blocks are inspected for functional coherence.
- *Block 3:* Figures 4 (f) shows the subsequent contraction of block at 3 and chain of *define-association* statements. The final graph in the figure shows three clusters/segments. The cluster classified as extracted method opportunity is represented by vertex with label *"1-13*"* represents cluster of statements from index 1 to 13, which corresponds to source code statements responsible for computing *n*th Fibonacci term.

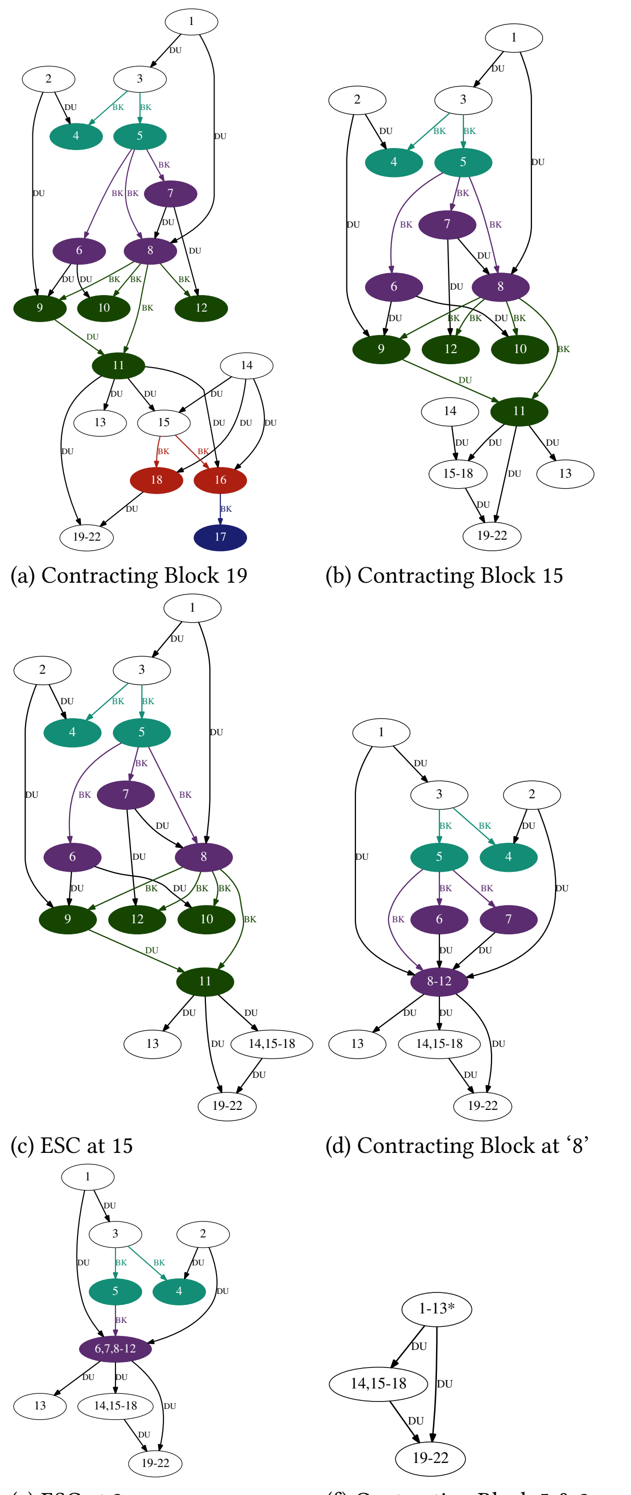

(a) Contracting Block 19
(b) Contracting Block 15
(c) ESC at 15
(d) Contracting Block at '8'
(e) ESC at 8
(f) Contracting Block 5 & 3

Figure 4: Structure Based Refactoring using Segmentation

3 EVALUATION

In this section, we present and discuss performance of the Segmentation approach along with other state-of-the-art approaches/tools, which include JExtract [18] and SEMI [5]. Here, the goal is to measure the accuracy of the approaches for identifying EMOs in six Java-based softwares, and also measure their scalability over long methods [14] (methods with 30 or more LoC).

3.1 Experimental Setup

The aforementioned approaches (JExtract, SEMI, and Segmentation) were applied to six Java-based open-source softwares (OSS) namely *JUnit*, *JHotDraw*, *MyWebMarket*, *EventBus*, *Mockito* and *XData*. Three of the six chosen softwares, *JUnit*, *JHotDraw* and *MyWebMarket*, have been used earlier in the literature [18], [5] as benchmarks for identifying EMOs. These benchmarks were created by silva et al. [18] by application of inline method refactoring. Using the same guidelines, we have added *EventBus* and *Mockito* to the dataset for evaluation of this study. The final addition is an OSS project called *XData* represents a grading system for queries [4]. Using this codebase strengthens our experimentation as it contains methods of varied lengths and of larger size as compared to those in the other two softwares. XData contains methods of sizes between 33-299 LoC. Further, four of these contain 200+ statements. We note that all studies contain exactly one premarked EMO per method, except XData and MyWebMarket.

3.2 Performance Metrics Used for Comparison

To measure the performance we use precision, recall and F measure metrics. Extract method opportunities listed by our approach and other state-of-the-art tools are compared with pre-marked opportunities, and the shares of True Positives, False Positives, and False Negatives are identified as mentioned below.

- *True Positive (TP):* opportunity identified by both the developer and the algorithm
- *False Positive (FP):* opportunity identified only by the algorithm
- *False Negative (FN):* opportunity identified only by the developer

$$Precision = \frac{\#TPs}{Retrieved\ opportunities} = \frac{\#TPs}{(\#TPs + \#FPs)}$$

$$Recall = \frac{\#TPs}{Relevant\ opportunities} = \frac{\#TPs}{(\#TPs + \#FNs)}$$

$$Fmeasure = \frac{2*(Precision*Recall)}{(Precision+Recall)}$$

Match Tolerance: We have analyzed the effect of applying *match tolerance* values of 1 to 3 statements on the identification of opportunities by our algorithm. A deviation of a match within the tolerance range in both direction is acceptable. The same tolerance value is used for all the tools used in the study. Absolute tolerance is preferred over percentage due to the observation that despite the presence of large methods (up to 299 LoC) in the studies, most of the EMOs are around 20 LoC and only a couple of outliers were observed beyond 31 LoC.

3.3 Data Collection and Tool Configuration

Murphy-Hill et al. [17] noted that 90% of users do not change the default configuration of a refactoring tool. So, we aim to keep the number of configurable parameters low; in our case, it is three. We also obtained default values for them through these four case studies. For evaluation, we collected the following information for selected Java-based case studies.

- *Size* we used JExtract [18] for measuring size of methods.
- *Golden Standard* JUnit, JHotDraw and MyWebMarket projects provide already marked extract method opportunities, whereas, for the XData grading system, we requested one of the developer (a research scholar with some industry experience) to identify the extract method opportunities manually. The developer marked extract method opportunities in the XData system. Finally, for EventBus and Mockito, statements of inlined methods were marked as extract method opportunities, similar to JUnit and JHotDraw. These markings were considered as the reference for evaluating the proposed approach.
- *Tool's EMO Suggestions* As discussed earlier, JExtract and SEMI generate an exhaustive set of suggestions. Hence, we follow the same criteria that was used for evaluation of SEMI [5], and restrict the suggestion count to 3*E suggestions per method, where E is the count of EMOs present in the method under consideration for refactoring.

3.4 The Process of Tuning

The proposed approach primarily uses two parameters to control the number of extract method opportunities (suggestions), namely, *LoCS*, *PA* as discussed earlier. To find the appropriate threshold values for the parameters a two part study is performed. In first part, we performed a study on an older version of segmentation approach and analyzed the extract method suggestions against varying thresholds for LoCS and PA. Next, the obtained threshold were applied to a selected group of methods of varying sizes from three OSSs, to reaffirm the performance of obtained threshold values. The values of LoCS and PA found in the case study discussed in this subsection also provide encouraging results for six other case studies. Now, we discuss the method of tuning LoCS and PA.

(1) An older *modularized base code* of segmentation implemented in C was first transformed into *non-modularized code*, by *unfolding* a few functionalities into their caller functions. New functions of varying sizes were formed by unfolding functions in the source. The non-modularized code for our evaluation that is generated from the above method consisted of a total of 28 distinct functionalities spread over 13 functions. Ten out of these 13 functions contain a single distinct functionality, resulting in three large functions containing 18 functionalities all in all. The original modularized base code was considered for comparing the results. Table 2 shows the precision and recall values for the transformed code. It is observed that threshold values of 0.41 and 0.34 respectively for LoCS and PA, provided the best results for this case study.

(2) Metric values obtained from the first study are again applied to 23 methods selected from three OSSs. These methods were selected from varying ranges of method size. From each OSS two candidate methods were selected belonging to the following size ranges: [1,10], [11,20], [21,30), and [30,30+]. Results obtained for three threshold values of LoCS metric are shown in Table 3. It can be observed that LoCS threshold 0.41 outperforms other thresholds in recall and f-measure with comparable precision.

So, threshold values of 0.41 and 0.34 for LoCS and PA, respectively, are used as default parameter settings for segmentation, when applied over six OSSs for evaluation of the proposed approach. It can be noted that the parameters are open to tuning from user as per the need of the application.

Table 2: Precision and Recall obtained after tuning

Methods to be refactored	#LoC	TP	FP	FN	Precision	Recall
#1 FindExcSourceNodes	55	3	1	2	0.75	0.60
#2 InitializeSourceList	23	2	1	0	0.66	1.00
#3 FindChain	137	5	3	6	0.62	0.54

Table 3: Performance for varying LoCS thresholds

Threshold	Precision	Recall	F-measure
0.37	16, 24, 32	17.4, 26.1, 34.8	16.7, 25, 33.3
0.41	20, 26.7, 30	26.1, 34.8, 39.1	22.6, 30.2, 34
0.45	16, 28, 32	17.4, 30.4, 34.8	16.7, 29.2, 33.3

3.5 Application to OSS Code: Results and Discussion

In this section, we present results obtained by applying all three approaches over six Java-based open source softwares. Tool's suggestions and performance for each OSS, individually and combined, is presented in Tables 4 and 5. Further, performance of the approaches over long methods, methods with 30 LoC or more [14], is illustrated in Table 6. The results are now analyzed with pointers to future work.

- *Nature and impact of suggestions:* It can be observed from Table 4 that Segmentation has provided significantly fewer suggestions compared to the other two approaches. Despite restricting the count of suggestions for JExtract and SEMI, they have provided 2.81 and 1.67 suggestions per EMO, respectively. Whereas the same rate for Segmentation is 1.07, almost one suggestion per EMO. Thus, it can be concluded

Table 4: EMO Suggestions Statistics of Tools/Approaches

OSS/Tools	JExtract	SEMI	Segmentation
JUnit (25)	72	39	17
JHotDraw (56)	133	130	67
MyWebMarket (35)	105	89	9
EventBus (24)	71	59	27
Mockito (46)	126	95	50
XData (110)	327	83	148
Total (296)	834	495	318

Table 5: Comparison for All Studies Combined

Tools	Tolerance	Precision	Recall	F measure
JExtract	1	18.75	52.86	27.68
	2	20.19	56.90	29.80
	3	20.91	58.92	30.86
SEMI	1	17.37	28.95	21.72
	2	23.63	39.39	29.54
	3	26.26	43.77	32.82
Segmentation	1	22.81	24.58	23.66
	2	30.	32.32	31.11
	3	38.75	41.75	40.19

Table 6: Performance over long methods (LOC>30)

Tools	Tolerance	Precision	Recall	F measure
JExtract	1	9.42	19.00	09.60
	2	6.98	20.66	10.44
	3	7.26	21.48	10.85
SEMI	1	16.35	14.04	15.11
	2	20.19	17.35	18.67
	3	23.07	19.83	21.33
Segmentation	1	30.82	40.49	35.00
	2	33.33	43.80	37.86
	3	37.10	48.76	42.14

that using a selective similarity measure for clustering and disjoint suggestions effectively restricts massive suggestion generation.

- *Performance for all studies:* Table 5 shows results for all case studies combined. Below, we discuss the performances of all approaches in detail.
  - *precision* The impact of suggestion count can be observed on the precision of the respective approaches. JExtract with the highest number of suggestion count has been outperformed by SEMI and Segmentation, whereas, Segmentation has outperformed both the approaches. It shows that the presented approach has successfully identified the potential EMOs despite restricting the count of suggestions per method. Further, an increase in precision with higher tolerance is an encouraging result and indicates scope for further refinement in the approach.
  - *recall* JExtract has once again proven to be the best in terms of recall. At the same time, SEMI secured the second spot by staying ahead of Segmentation with a 2-7% margin.
  - *f measure* Given that no single approach has been outstanding in both precision and recall, performance in terms of F measure was expected to be mixed. For tolerance 1, JExtract leads by a margin of 4% in the table. However, for tolerance 3, Segmentation outperforms both approaches by 7
  - *Conclusion* The high precision and F-measure achieved by Segmentation with almost one suggestion per EMO indicates that the approaches can achieve a higher performance without generating massive suggestions.
- *Performance over long methods :* One of the crucial requirements for an approach suggesting EMOs is their scalability over long methods. To measure the performance of all the approaches over long methods, we identified methods with 30 or more lines of code. Table 6 tabulates the results for the segmentation approach. The segmentation provided significantly better performance than all the other approaches. The second-best performance in terms of precision and F-measure was observed for SEMI, and JExtract in terms of recall.
- *Implications and future works:* The findings reveal that the performance of existing approaches can be improved by reducing the count of suggestions. This could be achieved by methods such as applying restricted configurations, filtering of suggestions that have heavy overlaps, and suggesting distinct suggestions.

3.6 Tool

Segmentation is implemented as an Eclipse plug-in. It is available as a jar[1] file. To use the plug-in, place it in 'plugins' directory of Eclipse IDE and restart (if already running).

4 RELATED WORK

Various restructuring approaches for Long Method code smell use techniques such as clustering [25] [3], control flow graphs [10, 11, 13] and program slicing [1], [8]. Lakhotia and Deprez [13] present an approach for restructuring by tucking sets of statements into functions. It separates two interleaved tasks into functions. This transformation takes place in three steps called *Wedge*, *Split* and *Fold*.

Lung and Zaman [15] present a clustering approach which exploits a variable's role in a statement to determine the type of data and control dependence associated with the statement. Conditional and iterative keywords (e.g. if, else, for, while) are also used as attributes of statements to compute similarity for clustering. Xu et al. [25] present a restructuring approach using clustering based on their resemblance coefficient, and observe desired clusters with data and control attribute ratios 5:2 and 8:3. Alkhalid et al. [3] present an approach called Adaptive-KNN which requires less computational time as their algorithm computes similarity matrix only once. Also, they note that results obtained for k=3 and k=5 are the same. So, the approach prefers clustering with k=3 because it further reduces the number of computations.

Program slicing was proposed by Weiser to help in software debugging by extracting a set of statements affecting computation of values of a set of variables at a specific program point[23]. Program slicing has found its place in many different software related activities such as debugging [2] [24], program comprehension [12], also restructuring [1, 16]. Abadi et al. [1] presented an approach to extract an executable slice corresponding to a *seed statement* with the help of external input. The external input includes data and control dependencies to be excluded. The extracted slice is transformed into a function around the seed statement and its closely related statements.

[1]https://www.cse.iitb.ac.in/~omkarendra/#Resources

Yal et al. [26] propose an approach for identifying candidate methods for extraction based on control structures, branches, blank lines etc. The approach also suggests that the size of the candidates for extraction should be above a threshold. The candidates for extraction are ranked using coupling computation. Silva et al. [18] propose an approach for generating exhaustive list of extract method candidates and then ranks them on the basis of similarity between the candidate and the remaining method statements.

Tsantalis and Chatzigeorgiou [21] present an approach based on complete computational slice that identifies the seed statements and variables without human involvement and suggests the extraction opportunity of a variable by computing union of the slices based on assignment, which are computed over all blocks for a chosen seed variable. Charalampidou et al. [5] present an approach to identify EMOs by clustering statements based on similarity on of identifiers present among them, then forming larger clusters by merging smaller ones. Further, cohesion metrics and overlapping is utilized to rank the relevant clusters as primary and alternative EMO suggestions.

4.1 Comparison with Related Work

In this section, we discuss three state-of-the-art approaches with tool support for extract method refactoring for identifying extract method refactorings and compare them with our approach.

The slicing based approach of JDeodorant [21] suggests EMOs centered around statements assigning value to seed variables in backward slice from the assignment statement. Such an approach limits its ability to identify large functionalities, as often such functionalities consist of computations dependent on the seed variable's computed value. This is not the case with approaches such as JExtract [18] and SEMI [5]. These approaches utilize identifiers among statements to cluster and suggest EMOs and outperform JDeodorant. However, these approaches generate a large number of suggestions, which provide higher recall but affect the precision. Furthermore, due to heavy overlapping among suggestions generated by the approaches multiple suggestions may point to the same functional blocks.

On the other hand, our approach of identifying extract method opportunities through Segmentation restricts the number of suggestions by identifying disjoint EMOs, which are computed by analyzing data association among statements in a block. The disjoint nature of suggestions allows the users to infer further insights about the structural complexity of a method under consideration. It can be used to prioritize methods for refactoring. Furthermore, to our knowledge, this is the first comparative study to be evaluated over Open Source Software containing multiple methods of size 100 to 299 LoC. It can be noted that SEMI [5] has been evaluated two industrial methods of size 500 lines of code each(approximately).

5 CONCLUSIONS

Segmentation, a novel graph-based approach for extract method refactoring, was presented. The approach suggests disjoint functionalities, resulting in fewer suggestions with a higher precision and a comparable recall. The approach was evaluated on code from six open-source softwares. The findings suggest that segmentation performs well for both small and long sized methods. Also, the suggestions can be utilized to infer further insight about methods to prioritize the refactoring order. The segment graph based framework developed in the paper is open to tuning. The approach can be used as a *refactoring assistant* to aid human experts in identifying modular functionalities through extract method opportunities.

REFERENCES

[1] Aharon Abadi, Ran Ettinger, and Yishai A Feldman. 2012. Fine Slicing. In *Fundamental Approaches to Software Engineering*. Springer, 471–485.

[2] Hiralal Agrawal, Richard A DeMillo, and Eugene H Spafford. 1993. Debugging with dynamic slicing and backtracking. *Software: Practice and Experience* 23, 6 (1993), 589–616.

[3] Abdulaziz Alkhalid, Mohammad Alshayeb, and Sabri Mahmoud. 2010. Software refactoring at the function level using new Adaptive K-Nearest Neighbor algorithm. *Advances in Engineering Software* 41, 10 (2010), 1160–1178.

[4] Amol Bhangdiya, Bikash Chandra, Biplab Kar, Bharath Radhakrishnan, KV Maheshwara Reddy, Shetal Shah, and S Sudarshan. 2015. The XDa-TA system for automated grading of SQL query assignments. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*. IEEE, 1468–1471.

[5] Sofia Charalampidou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, Antonios Gkortzis, and Paris Avgeriou. 2017. Identifying Extract Method Refactoring Opportunities based on Functional Relevance. *IEEE Trans. Software Eng.* 43, 10 (2017), 954–974.

[6] Martin Fowler. 2009. *Refactoring: improving the design of existing code.* Pearson Education India.

[7] Jonathan L Gross, Jay Yellen, and Mark Anderson. 2018. *Graph theory and its applications.* Chapman and Hall/CRC.

[8] Hyeon Soo Kim, Yong Rae Kwon, and In Sang Chung. 1994. Restructuring programs through program slicing. *International Journal of Software Engineering and Knowledge Engineering* 4, 03 (1994), 349–368.

[9] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2012. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 50.

[10] Raghavan Komondoor and Susan Horwitz. 2000. Semantics-preserving procedure extraction. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 155–169.

[11] Raghavan Komondoor and Susan Horwitz. 2003. Effective, automatic procedure extraction. In *11th IEEE International Workshop on Program Comprehension, 2003.* IEEE, 33–42.

[12] Bogdan Korel and Juergen Rilling. 1998. Program slicing in understanding of large programs. In *Program Comprehension, 1998. IWPC'98. Proceedings., 6th International Workshop on*. IEEE, 145–152.

[13] Arun Lakhotia and Jean-Christophe Deprez. 1998. Restructuring programs by tucking statements into functions. *Information and Software Technology* 40, 11 (1998), 677–689.

[14] Martin Lippert and Stephen Roock. 2006. Refactoring in Large Software Projects.

[15] Chung-Horng Lung and Marzia Zaman. 2004. Using Clustering Technique to Restructure Programs.. In *Software Engineering Research and Practice*. 853–860.

[16] Katsuhisa Maruyama. 2001. Automated method-extraction refactoring by using block-based slicing. In *Proceedings of the 2001 symposium on Software reusability: putting software reuse in context*. 31–40.

[17] Emerson R. Murphy-Hill, Chris Parnin, and Andrew P. Black. 2012. How We Refactor, and How We Know It. *IEEE Trans. Software Eng.* 38, 1 (2012), 5–18. https://doi.org/10.1109/TSE.2011.41

[18] Danilo Silva, Ricardo Terra, and Marco Tulio Valente. 2014. Recommending automated extract method refactorings. In *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, 146–156.

[19] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 858–870.

[20] Omkarendra Tiwari and Rushikesh K Joshi. 2020. Functionality Based Code Smell Detection and Severity Classification. In *Proceedings of the 13th Innovations in Software Engineering Conference on Formerly known as India Software Engineering Conference*. 1–5.

[21] Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2011. Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software* 84, 10 (2011), 1757–1782.

[22] Carmine Vassallo, Giovanni Grano, Fabio Palomba, Harald C Gall, and Alberto Bacchelli. 2019. A large-scale empirical exploration on refactoring activities in open source software projects. *Science of Computer Programming* 180 (2019), 1–15.

[23] Mark Weiser. 1981. Program slicing. In *Proceedings of the 5th international conference on Software engineering*. IEEE Press, 439–449.

[24] Baowen Xu, Zhenqiang Chen, and Hongji Yang. 2002. Dynamic slicing object-oriented programs for debugging. In *Source Code Analysis and Manipulation, 2002. Proceedings. Second IEEE International Workshop on*. IEEE, 115–122.

[25] Xia Xu, Chung-Horng Lung, Marzia Zaman, and Anand Srinivasan. 2004. Program restructuring through clustering techniques. In *Source Code Analysis and Manipulation, 2004. Fourth IEEE International Workshop on*. IEEE, 75–84.

[26] Limei Yang, Hui Liu, and Zhendong Niu. 2009. Identifying fragments to be extracted from long methods. In *Software Engineering Conference, 2009. APSEC'09. Asia-Pacific*. IEEE, 43–49.